Pattern Mining for Future Attacks

Sandeep Karanth
Microsoft Research India
skaranth@microsoft.com

Srivatsan Laxman
Microsoft Research India
slaxman@microsoft.com

Prasad Naldurg
Microsoft Research India
prasadn@microsoft.com

Ramarathnam Venkatesan
Microsoft Research India
venkie@microsoft.com

J. Lambert
Microsoft Corporation

Jinwook Shin
Microsoft Corporation
jinshin@microsoft.com

ABSTRACT

Malware writers are constantly looking for new vulnerabilities in popular software applications to exploit for profit, and discovering such a flaw is literally equivalent to finding a gold mine. When a completely new vulnerability is found, and turned into what are called Zero Day attacks, they can often be critical and lead to data loss or breach of privacy. Zero Day vulnerabilities, by their very nature are notoriously hard to find, and the odds seem to be stacked in favour of the attacker. However, before a Zero Day attack is discovered, attackers stealthily test different payload delivery methods and their obfuscated variants, in an attempt to outsmart anti-malware protection, with varying degrees of success. Evidence of such failed attempts, if any, are available on the victim machines, and the challenge is to discover their signatures before they can be turned into exploits. Our goal in this paper is to search for such vulnerabilities and straighten the odds. We focus on Javascript files, and using a combination of pattern mining and learning, effectively find two new Zero Day vulnerabilities in Microsoft Internet Explorer, using code collected between June and November 2009.

1. INTRODUCTION

Consider a malware-writer who wants to test a new vulnerability and turn it into an attack. In order to deliver the attack payload to a victim machine, the attacker could use a poorly validated Javascript form on a third-party web application, e.g., by using "eval" in string input, and redirect the scripting engine to fetch and execute their code. Increasingly, Javascript is being exploited as a vehicle to mount reflected attacks (including XSS attacks) of this nature. As part of the attack, buffer overflow attempts, heap overflow attempts and heap sprays are inserted into the current execution context, in the hope that they succeed in subverting control to execute attacker-inserted code. Attack payload has also been found in Javascript snippets in documents such as Adobe PDF files. The SANS 2009 report lists exploitation of client-side vulnerabilities using vulnerable Internet-facing websites as the most important unmitigated cybersecurity risk [16], forming over 60% of reported attacks, with web-application vulnerabilities resulting in reflected attacks accounting for over 80% of such vulnerabilities discovered.

Static analysis techniques [3, 10, 4, 9, 6] can identify attack payload in many cases, when the code is available ahead of time. However, this is not a complete solution for web applications that include dynamic content, such as context-sensitive advertisements. Imposing controls on how and where dynamic code can be downloaded on a form may be restrictive in practice, as only a small percentage of the downloaded code is malicious (0.06% based on our own study, Section 4).

In order to mount a successful attack, a malware-writer will need to prevent the detection of the attack payload by any protective software deployed on a victim (Note that the actual vulnerability itself will be new, and unknown to any signature scanners). It is difficult in general for an attacker to write attack code that uses delivery methods that are substantially different from those that have worked in the past. To work around this, a typical attacker uses a variety of code-obfuscation techniques, such as introducing intermediate variables to store string constants, removing literals, encryption etc., with varying degrees of success. Before a successful attack is developed, attackers may test a carefully crafted version of the payload on a target configuration. This attempt may fail, but will leave the attack fingerprints behind on the victim machine. If these fingerprints can be found, they can provide valuable information about potentially dangerous zero-day vulnerabilities (ZDV), which are undisclosed to a software developer, for which no security patches or anti-malware protections are available. Based on our own observations, these events occur at extremely low frequencies, as attackers are careful about releasing them widely before they work, and attackers may even wait for months before the same vulnerability is tried again, using a different delivery mechanism.

Increasingly, malware writers are being hired by criminal organizations with vested interests to indulge in targeted cybercrime [5]. In this market, ZDVs have a high premium, fetching upto $250000 in 2007 [12], and a consortium of industry and researchers [13] are even attempting to legitimize the disclosure of Zero Days with economic incentives. High value ZDVs have the potential to cause significant damage, leading to data loss, data theft, data corruption and manipulation and privacy breaches. As a leveraging tool for a cyber criminal, they are only valuable as long as the par-

ticular vulnerability remains unpatched, and unknown to everybody except the attackers. In this context, finding attempted ZDVs before they can be exploited becomes increasingly important.

As mentioned earlier, in practice, only a small percentage of downloaded Javascript is malicious. Further, malicious payload shares many common features with innocuous dynamic code, and the challenge is to systematically separate the two, and characterize and identify malicious Javascript code to find ZDVs. We approach this task with the intuition that it is possible to characterize the difference between malicious and benign Javascript. The difference may be small, but sophisticated statistical techniques can help magnify this difference and identify malicious payload with a high degree of confidence.

To this end, we construct an alphabet of features or indicators (and some of their obfuscated variants) that correspond to Javascript codewords and functions that occur frequently only in malicious payload, features that occur mostly in benign payload, and features that occur in both. Once we have an alphabet, we estimate the co-occurrence statistics, i.e., the frequency of these features as patterns over this alphabet. We use annotated data from a small sample of known malicious Javascript files, as well as a larger sample of benign files, to learn a set of weights for these pattern-frequencies. With these weights, we develop a scoring technique that discriminates malicious payload from benign effectively, and prioritizes them based on their likelihoods in each class. Using these patterns and weights, we test our algorithm on hundreds of thousands of unannotated Javascript files collected over a period of 4 months, from heterogeneous sources, and score each of these reports accordingly.

Our main contribution is an automated prioritization scheme for identifying malicious Javascripts. We develop a scoring technique for Javascript that uses co-occurrence statistics of features to assign a likelihood score to each Javascript file. Our prioritizer ranks the output list of Javascript files according to these scores; suspicious cases with potential of being exploit-attempts on new ZDVs are assigned higher scores (and are pushed-up the ranked list). Our algorithm alleviates the massive manual effort required of security experts to detect ZDV exploit-attempts from a large number of Javascript files (that pass through standard anti-malware signature scanners). For example, our algorithm surfaced a list of 384 new vulnerability candidates that were unknown to our anti-malware signature scanners, out of a total of 26932 code files. Importantly, we were able to detect a new ZDV (unknown at the time of data collection and testing, but disclosed subsequently) and an obfuscated variant of the ZDV; these were reported within the top 1% and 2% respectively of our prioritized reports. We demonstrate good precision and recall performance on annotated data. Note that, when completely new attacks (using substantially different payload delivery techniques) surface, we introduce new features and retrain our model. The techniques developed in this paper have been incorporated in a diagnostic tool which is used on a day-to-day basis by security experts at Microsoft.

The rest of the paper is organized as follows: Section 2 describes the characteristics of our dataset, with a special emphasis on our selected features and presents our intuition for using co-occurrence statistics. In Section 3 we describe our method in detail, concentrating on both the training and testing phases. In Section 4 we describe our data collection and the results of applying our method to this dataset, along with a sensitivity study. Section 5 presents related research, followed by conclusions and future work in Section 6.

2. BACKGROUND

In this section, we explain our intuiton for using the co-occurence of features as a discriminatory basis for classifying Javascript payload. We collect Javascript internet browser attack data, obtained from Microsoft customer reported incidents, submissions of malicious code and Windows error reports, between July and November 2009. We start with hundreds of thousands of Javascript code samples, and apply a filter to identify Javascript files that are interesting from a security perspective, using a list of code features or findings. These features are selected intuitively to represent: (i) a mix of commonly occuring actions in interesting Javascript code (both in benign and malicious), (ii) features that largely occur only in benign code, and (iii) features that are largely indicators of malicious attack payload. A Javascript 'transaction' is a snippet of Javascript code that contains one or more findings. Each transaction represents around 20 lines of Javascript code. After filtering, around 130000 transactions were collected in the period of July-November 2009.

The Javascript features are listed in Table 1. For example, in a typical heapspray attempt, an attacker is likely to use feature 28 (a new_array call to allocate memory, which is a feature of any Javascript which requires dynamic storage) along with features 20 and 14 (unescape to decode encoded strings, and NOPs). Our choice for three kinds of features is motivated by our observation that using only features that occur frequently in malicious code snippets leads to detection of a lot of false positives, as benign code also contains many of these features. Similarly, using only the subset of features in the malicious list that do not occur (or occur with low frequencies) in the benign samples leads to substantially many false negatives, often all-together missing the ZDV attempts. A mix of features allows a more robust statistical characterization of Javascript files. Table 2 shows a subset of 7 commonly occurring features in a Javascript attack that tries to exploit three different known vulnerabities in the Microsoft Internet Explorer browser, prior to October 2009 (Source: National Vulnerability Database, Common Vulnerabilites and Exposures (CVE) with identifiers, CVE-2008-4844, CVE-2008-0015 and CVE-2009-0075). A code snippet for this attack, with the some features highlighted is provided in Fig. 1.

Table 2 describes the marginal statistics of the selected features in malicious and benign code. Note that some of these features occur exclusively in malicious samples: feature 14, which indicates injection of NOP-sled blocks, and feature 17, which indicates shell code injection. However, both their frequencies are only about 60%; so if we restricted our classifier to only these features we would miss around 40% of the attacks. That said, whenever 14 and/or 17 occur(s), the Javascript transaction should be given a high score (marking it as highly suspicious). On the other hand, feature 23 (that indicates definiton of a Javascript class or function) and feature 18 (call to a substring function) both have comparable frequencies in malicious and benign samples; hence they do not, by themselves, add much value for discriminating malicious cases from benign. Feature 28 (which represent allocation of memory) and feature 20 (a

Id	Feature	Description	Id	Feature	Description
1	document.write	Write HTML expression to document	18	substr	Extracts characters from a string
2	evaluate	Evaluates and/or executes a string	19	shellcode	Shell code presence in JS string
3	push	Add array elements	20	unescape	Decode an encoded string
4	msDataSourceObject	ActiveX object for Microsoft web components	21	*u0A0A_u0A0A*	Injection of characters 0A0A
5	setTimeout	Evaluates expression after timeout	22	CompressedPath	Download path property on an ActiveXObject
6	cloneNode	Create object copy	23	function	Function or class definition
7	createElement	Create HTML element	24	shellexecute	Call Shell API
8	window	Current Document object	25	*u0c0c_u0c0c*	Injection of characters 0c0c
9	getElementbyId	Get an element from the current document	26	replace	Replace a matched substring with a string
10	object_Error	JS exception object	27	Collab.CollectEmailInfo	Adobe Acrobat method for for email details
11	*u0b0c_u0b0b*	Injection of character 0b0c0b0b	28	new_Array	Heap memory allocation
12	CollectGarbage	Call to garbage collector	29	*u0b0c_u0b0c*	Injection of characters 0b0c0b0c
13	SnapshotPath	Snapshot path property on an ActiveXObject	30	LoadPage	Load a HTML expression
14	*u9090_u9090*	NOP injection	31	createControlRange	Create a control container
15	new_ActivexObject	Instantiate ActiveX object	32	Click	JS click event
16	navigator.appversion	Get browser version	33	appViewerVersion	Get Adobe Acrobat Reader version
17	0x0*c*0*c*0*c*0*c*	Injection of 0c0c0c	34	function_packed	JS packer function

Table 1: Javascript Features

Id	Feature	% Count in malicious snippets	% Count in benign snippets	% Difference
14	*u*9090_*u*9090	61	0	61
17	0x0*c*0*c*0*c*0*c*	58	0	51
18	substr	95	88	7
20	unescape	85	45	40
23	function	97	99	2
26	replace	64	87	23
28	new_array	89	48	41

Table 2: Marginal statistics of some selected features found in CVE- 2008-4844, CVE-2008-0015 and CVE-2009-0075.

call to decode an encoded string) both occur in large numbers of malicious samples ($> 85\%$) but also occur in close to 50% of benign samples; hence, while these features have potential as discriminators of malicious cases from benign, when used alone, they can lead to a substantial false positive rates. Similarly, feature 26, can be a useful discriminatory feature for benign code, but when used alone, it can result in a high false negative rate.

The challenge is to build a classifier that can automatically select 'interesting' features to disambiguate malicious scripts from benign ones. In this paper, we develop statistical methods to identify *significant* 'groups of features' in malicious and benign code-snippets. The essence of our significance test is to determine whether some groups of features occur with substantial statistical skew in the malicious examples as compared to the benign ones. Typically, these skews are easier to detect in higher order-statistics than in single (or marginal) feature statistics, and hence we use patterns of frequently co-occuring features rather than individual features. We illustrate this with an example. Table 3 presents the co-occurence statistics of various combinations of features 20 - unescape, 26 - replace, 28 - new_array. Note that the combination of features 20 and 28 is a stronger attack indicator (55% difference) than if they were considered individually ($40 - 41\%$ difference). Combined with other indicators of known heap sprays like nopsleds and shellcode, e.g., features 14 and 17, these become stronger indicators of malicious Javascripts. Using multiple features also builds in robustness to new forms of obfuscation that may not be directly recognized by exact signature scanners. Further, a combination of all three features $(20, 26, 28)$, for example, has a lower occurence in positive samples than any pair of features, highlighting the need for a weighted scoring technique to automatically identify high-value indicators. In this paper, we employ a scoring technique that converts the observed co-occurrence statistics into likelihood scores using a data set of annotated examples (marked benign or malicious by a security expert).

As a final example, we examine a new zero-day vulnerability that surfaced on November 29th 2009 (CVE-2009-3672). The code for the vulnerability is given in Figure 2. This particular attack can be encoded using features 18, 20, 23 and 28. We show later in Section 4 that we were able to detect this exploit using our method based on a weighted combination of features. We explain our classification method in detail in Section 3 next.

3. METHOD

Our goal is to identify high-value malicious Javascript payload from our collected samples and look for future attack signatures. The overall approach consists of a training phase and a ranking phase. In the training phase, we use a set of

Figure 1: Example of exploit code for three vulnerabilities CVE- 2008-4844, CVE-2008-0015 and CVE-2009-0075 (with some features highlighted).

Pattern	Description	% Count in malicious snippets	% Count in benign snippets	% Difference
20, 26, 28	unescape, replace, new_array	49	23	26
20, 26	unescape, replace	51	37	14
20, 28	unescape, new_array	83	28	55
26, 28	replace, new_array	54	40	14

Table 3: Co-occurrence (or higher-order) statistics of features found in CVE- 2008-4844, CVE-2008-0015 and CVE-2009-0075.

annotated transactions to learn a probabilistic model for the data (*Algorithm 1*). In the ranking phase, we prioritize the unannotated transactions based on their likelihood scores with respect to the learnt model (*Algorithm 2*). This section presents details of the training and ranking phases in our method.

We begin with an overview of the general procedure. For the training phase, we are given annotated sets of positive transactions (S_+) and negative transactions (S_-). As mentioned earlier, each transaction corresponds to a set of findings in a snippet of Javascript code. It is is essentially a collection of *items* (or symbols) over a finite alphabet (say $\mathcal{A}$). In our context, the alphabet $\mathcal{A}$ is the universal set of findings or features which are identified as relevant for disambiguating malicious scripts from benign ones. These features were selected from our annotated samples statistically.

An *itemset* is a non-empty set of items, e.g., $\alpha = (A_1 \ldots, A_N)$, $A_i \in \mathcal{A}$, $i = 1, \ldots, N$ denotes an N-itemset or an itemset of size N. Given a collection of transactions (say $\mathcal{D}$) the frequency of α in $\mathcal{D}$ is the number of transactions that contain (or support) α. Patterns whose frequency exceeds a user-defined threshold are referred to as *frequent itemsets*. In our context, a frequent itemset is simply a group of features that co-occur frequently in the Javascript snippets. The goal of the training phase is to estimate which groups of features correlate strongly in the positive examples but are absent (or are weakly correlated) in the negative examples, and vice versa. Based on these, we build separate probabilistic models for the positive and negative examples. During the ranking phase, we pass each test point through these models, obtain likelihood scores for the point (under the positive and negative models) and calculate the corresponding likelihood ratio (positive score over negative score). We then rank the test points in decreasing order of likelihood ratios.

The main steps in the training phase are listed in *Algorithm 1*. Input to the training phase are sets $\mathcal{S}_+$ and $\mathcal{S}_-$ of positive and negative training transactions, along with a user-defined maximum size N of patterns to be considered. Typical values we use in our experiments are $N = 5$ or $N = 6$. The first steps (lines 1, 2, *Algorithm 1*) compute the frequent itemsets for both data sets $\mathcal{S}_+$ and $\mathcal{S}_-$. The the task

Figure 2: Exploit code for ZDV CVE-2009-3672.

Algorithm 1 Training algorithm

Input: Data set of positive examples ($\mathcal{S}_+$), data set of negative examples ($\mathcal{S}_-$), maximum size of patterns (N)
Output: Generative models Λ_+ (for $\mathcal{S}_+$) and Λ_- (for $\mathcal{S}_-$)

1: Find the set ($\mathcal{F}_+$) of frequent itemsets in $\mathcal{S}_+$ of size N or less, using a frequency threshold of $\frac{|\mathcal{S}_+|}{2^N}$
2: Find the set ($\mathcal{F}_-$) of frequent itemsets in $\mathcal{S}_+$ of size N or less, using a frequency threshold of $\frac{|\mathcal{S}_-|}{2^N}$
3: Associate each α_j in $\mathcal{F}_+ \cup \mathcal{F}_-$ with an IGM Λ_{α_j} (based on *Definition 1*)
4: Keep only 'significant' patterns in $\mathcal{F}_+$ and $\mathcal{F}_-$ (based on Eq. 2)
5: Keep only 'discriminating' patterns in $\mathcal{F}_+$ and $\mathcal{F}_-$ by removing patterns common to both
6: Build a mixture (Λ_+) of significant IGMs (Λ_{α_j}, $\alpha_j \in \mathcal{F}_+$) for $\mathcal{S}_+$ (using Eqs. 3–5)
7: Build a mixture (Λ_-) of significant IGMs (Λ_{α_j}, $\alpha_j \in \mathcal{F}_-$) for $\mathcal{S}_-$ (using Eqs. 3–5)
8: Output Λ_+ and Λ_-

of discovering *all* itemsets whose frequencies exceed a given user-defined frequency threshold is a well-studied problem in data mining called Frequent Itemsets Mining (FIM) [1]. We use a standard procedure known as the Apriori algorithm for obtaining frequent itemsets in the data [1]. We choose the FIM framework for computing co-occurences, over computing joint probability estimates, as we do not want to make any assumptions on the underlying distributions of the findings, and efficient algorithms are available to compute these frequencies.

For each pair (itemset, frequency), we need a way of deciding if it is a useful discrimininating statistic. We use a significance test to decide whether an itemset is interesting or not, rejecting all patterns whose frequencies are below a noise threshold, which corresponds to the likelihood that the pattern occurs no frequently than background noise. In order to fix this threshold, we associate each frequent itemset discovered with a simple generative model called an Itemset Generating Model (or IGM) [7]. Using IGMs, we build a plausible generative model for the itemset frequencies that has the following properties:

1. The probability of a transaction is high whenever the transaction contains the itemset of interest.

2. Ordering of transaction frequencies is preserved under the data likelihoods suggested by their corresponding IGMs.

Note that our IGM model may not be the best generative model that fits the data, as we are only interested in thresholding itemsets whose frequencies are not statistically significant.

The main utility of the itemset-IGM associations of [7] is a test of significance for frequent itemsets in the data based on a likelihood ratio test involving IGMs. This is important because we only want to use patterns that are 'signficant' in the sense that they could not have appeared with such a frequency by random chance. We present the details of the IGMs and this connection next. As mentioned earlier, IGM Λ_α for itemset α is one under which the probability of a transaction T is high whenever T contains α. An Itemset Generating Model (or IGM) is specified by a pair, $\Lambda = (\alpha, \theta)$, where $\alpha \subseteq \mathcal{A}$ is an N-itemset, referred to as the "pattern" of Λ, and $\theta \in [0, 1]$ is referred to as the "pattern probability" of Λ. The class of all IGMs, obtained by considering all possible itemsets of size N (over $\mathcal{A}$) and by considering all possible pattern probability values $\theta \in [0, 1]$, is denoted by $\mathcal{I}$. The probability of generating a transaction T under the IGM Λ is prescribed as follows:

$$P[T \mid \Lambda] = \theta^{z_\alpha(T)} \left(\frac{1-\theta}{2^N - 1}\right)^{1-z_\alpha(T)} \left(\frac{1}{2^{M-N}}\right) \quad (1)$$

where, $z_\alpha(\cdot)$ indicates set containment: $z_\alpha(T) = 1$, if $\alpha \subseteq T$, and $z_\alpha(T) = 0$, otherwise; and M denotes the size of the alphabet $\mathcal{A}$. Observe that even for moderate values of θ (namely, for $\theta > \frac{1}{2^N}$) the above probability distribution peaks at transactions that contain α, and the distribution is uniformly low everywhere else. The *itemset-IGM association* is defined as follows:

DEFINITION 1. *[7] Consider an N-itemset, $\alpha = (A_1, \ldots, A_N)$ ($\alpha \in 2^{\mathcal{A}}$). Let f_α denote the frequency of α in the given database, $\mathcal{D}$, of K transactions. The itemset α is associated with the IGM, $\Lambda_\alpha = (\alpha, \theta_\alpha)$, with $\theta_\alpha = (\frac{f_\alpha}{K})$, if $(\frac{f_\alpha}{K}) > (\frac{1}{2^N})$, and with $\theta_\alpha = 0$ otherwise.*

In line 3, *Algorithm 1*, we associate each frequent itemset with a corresponding IGM as per the above definition. This itemset-IGM association has several interesting properties [7]. First, the association under *Definition 1* ensures that ordering with respect to frequencies among N-itemsets over $\mathcal{A}$ is preserved as ordering with respect data likelihoods among IGMs in $\mathcal{I}$. Further, if the most frequent itemset is 'frequent enough' then it is associated with an IGM which is a maximum likelihood estimate for the data, over the full class, $\mathcal{I}$, of IGMs. (We refer the reader to [7] for theoretical details of the model and its properties).

Using the IGM-itemset association, it is possible to construct a significance test based on the evidence of at least one reasonable model (namely the IGM) over that of a uniform random source. This is the next step in our training phase (line 4, *Algorithm 1*). For a given user-defined level ϵ of the test, an itemset α of size N with frequency f_α in a data set of K transactions, is declared *significant* if $f_\alpha > \Gamma$ where Γ is given by

$$\Gamma = \frac{K}{2^N} + \sqrt{\left(\frac{K}{2^N}\right)\left(1 - \frac{1}{2^N}\right)}\Phi^{-1}(1-\epsilon) \qquad (2)$$

where, $\Phi^{-1}(\cdot)$ denotes inverse of the cumulative distribution function (cdf) of the standard normal random variable. This is a standard error characterization for the standard normal random variable. For typical values of size ϵ of the test, size K of the given transactions data set and size N of the itemsets under consideration, the above expression tends to be dominated by $\frac{K}{2^N}$, and so, in the absence of any other information, we use this as the first threshold for significance to try in our algorithm. If the eventual model obtained is too weak (because of either too few or too many significant itemsets) we can always go back to the significance testing step and tune the set of significant episodes by selecting an appropriate value of ϵ.

Note that while IGMs are useful to assess the statistical significance of a given itemset, no single IGM can be used as a reliable generative model for the whole data. This is because, a typical data set, $\mathcal{D} = \{T_1, \ldots, T_K\}$, would contain not one, but several, significant itemsets. Each of these itemsets has an IGM associated with it as per *Definition 1*. We suppose that a mixture of such IGMs, rather than any single IGM, can be a good generative model for $\mathcal{D}$. Similar ideas were used in a the context of stream prediction using frequent episodes in [8].

The final step in our training phase is the estimation of a mixture of IGMs for each data set $\mathcal{S}_+$ and $\mathcal{S}_-$ (lines 6, 7, *Algorithm 1*). We can think of this step as a simple iterative procedure that assigns weights to the signficant patterns found in the data. Let $\mathcal{F}^s = \{\alpha_1, \ldots, \alpha_J\}$ denote a set of significant itemsets in the data, $\mathcal{D}$. Let Λ_{α_j} denote the IGM associated with α_j for $j = 1, \ldots, J$. Each sequence, $T_i \in \mathcal{D}$, is now assumed to be generated by a mixture of the IGMs, $\Lambda_{\alpha_j}, j = 1, \ldots, J$. Denoting the mixture of EGHs by Λ, and assuming that the K transactions in $\mathcal{D}$ are independent, the likelihood of $\mathcal{D}$ under the mixture model can be written as follows:

$$\begin{aligned} P[\mathcal{D} \mid \Lambda] &= \prod_{i=1}^{K} P[T_i \mid \Lambda] \\ &= \prod_{i=1}^{K} \left(\sum_{j=1}^{J} \phi_j P[T_i \mid \Lambda_{\alpha_j}] \right) \end{aligned} \qquad (3)$$

where $\phi_j, j = 1, \ldots, J$ are the mixture coefficients of Λ (with $\phi_j \in [0,1]\ \forall j$ and $\sum_{j=1}^{J} \phi_j = 1$). Each EGH, Λ_{α_j}, is fully characterized by the significant itemset, α_j, and its corresponding pattern probability parameter, θ_{α_j} (cf. *Definition 1*). Consequently, the only unknowns in the expression for likelihood under the mixture model are the mixture coefficients, $\phi_j, j = 1, \ldots, J$. We use the Expectation Maximization (EM) algorithm [2], to estimate the mixture coefficients of Λ from the data set, $\mathcal{D}$. Any other reweighting technique can also work equally well.

Let $\Phi^g = \{\phi_1^g, \ldots, \phi_J^g\}$ denote the *current guess* for the mixture coefficients being estimated. At the start of the EM procedure, ϕ^g is initialized uniformly, i.e. we set $\phi_j^g = \frac{1}{J}\ \forall j$. By regarding ϕ_j^g as the prior probability corresponding to the j^{th} mixture component, Λ_{α_j}, the posterior probability for the l^{th} mixture component, with respect to the i^{th} transaction, $T_i \in \mathcal{D}$, can be written using Bayes' Rule:

$$P[l \mid T_i, \Phi^g] = \frac{\phi_l^g P[T_i \mid \Lambda_{\alpha_l}]}{\sum_{j=1}^{J} \phi_j^g P[T_i \mid \Lambda_{\alpha_j}]} \qquad (4)$$

After computing $P[l \mid T_i, \Phi^g]$ for $l = 1, \ldots, J$ and $i = 1, \ldots, K$, using the current guess, Φ^g, we obtain a *revised estimate*, $\Phi^{new} = \{\phi_1^{new}, \ldots, \phi_J^{new}\}$, for the mixture coefficients, using the following update rule. For $l = 1, \ldots, J$, compute:

$$\phi_l^{new} = \frac{1}{K} \sum_{i=1}^{K} P[l \mid T_i, \Phi^g] \qquad (5)$$

The revised estimate, Φ^{new}, is used as the 'current guess', Φ^g, in the next iteration, and the procedure (namely, the computation of Eq. (4) followed by that of Eq. (5)) is repeated until convergence. We use the above procedure to estimate separate mixture models for $\mathcal{S}_+$ and $\mathcal{S}_-$. The resulting generative models, Λ_+ and Λ_-, are the final outputs of the training phase (line 8, *Algorithm 1*).

We now describe the ranking phase of our procedure. The main steps in the ranking phase are listed in *Algorithm 2*. Inputs to the ranking phase are the generative models Λ_+ and Λ_- obtained from the training phase and the set $\mathcal{D}$ of test transactions that need to be prioritized. The first step is to compute, for each test transaction $T \in \mathcal{D}$, the likelihoods under the positive and negative generative models (lines 1-3, *Algorithm 2*). Then we compute the corresponding likelihood ratio (line 4, *Algorithm 2*). Finally, we sort the test cases in decreasing order of likelihood ratios (line 5, *Algorithm 2*). This sorted list is the final output of our ranking procedure (line 6, *Algorithm 2*).

Algorithm 2 Ranking algorithm

Input: Set $\mathcal{D}$ of test cases, learnt mixture models from training phase (Λ_+ and Λ_-)
Output: Prioritized list of test cases ($\mathcal{D}$)

1: **for** each test case $T \in \mathcal{D}$ **do**
2: Compute likelihood $P[T \mid \Lambda_+]$ of T in Λ_+ (based on Eq. (1) for Λ_+)
3: Compute likelihood $P[T \mid \Lambda_-]$ of T in Λ_- (based on Eq. (1) for Λ_-)
4: Obtain the likelihood ratio for T: $\frac{P[T \mid \Lambda_+]}{P[T \mid \Lambda_-]}$
5: Sort test cases in $\mathcal{D}$ in descending order of likelihood ratios
6: Output sorted list $\mathcal{D}$

Note that the final prioritized list bubbles transactions that have likelihood high scores from the positive model and lower scores from the negative model. Transactions which are more similar to our annotated S_- will appear in the bottom of the list, and transactions that do not have distinguishing features will closer to the middle.

3.1 Output filtering and Analysis

After the ranking phase, we examine the Javascript files corresponding to the top ranked transactions using our signature scanners. This output filtering step removes any known vulnerabilities for consideration as potential ZDVs. We observe that our method is useful in characterizing our transactions as follows:

- After training, we can run our testing algorithm on the training set itself, to validate if our learnt model can recall all annotated transactions accurately. If any annotated transactions are classified otherwise (e.g., S_+ as S_-), an expert can inspect them and check if the samples are internally consistent. This is useful to check the correctness of the training set.

- At the next level, after we run our prioritization and classification algorithm on the test data, and run our signature scanners and any other anti-malware protection on the top results, the remaining top results can either be new ZDVs, or simply known signatures/attacks that are not in our database yet, and suggest update our anti-malware signature database. Any ZDVs found will be extremely useful in any case.

Finally, as noted earlier, our results are sensitive to the choice of findings exposed in our alphabet. A completely new method of delivering a ZDV, which bears no statistical similarities to any existing attack, will not be identified by our method. However, we expect to periodically update our set of findings, in response to trends observed in incident reports and advisories.

4. EVALUATION

In this section we present results from applying our methods to sample IE attack data obtained from customer reported incidents, submissions of malicious code and windows error reports collected between July and November 2009. We present the characteristics of the data observed, in Section 4.1, and describe our training and validation experiences in Section 4.2. We presents results on detecting ZDVs in our test data in Section 4.3 and follow it up with a discussion on the scope and limitations of our approach, and a comment on adversial manipulation in Section 4.4.

Data Set	Date Range '09	All cases	Malicious cases	Benign cases	Avg. size of transactions
Train	Jul-Oct	13543	839	12704	11
Test	Aug-Nov	26932	453	26479	6

Table 4: Data characteristics

4.1 Data Characteristics

The characteristics of the data used in our evaluation are listed in Table 4. The *Train Data* consists of 13500 snippets of Javascript code, which were filtered from a larger set of samples collected between July and October 2009. Similarly, the *Test Data*, which contains 26932 snippets of Javascript code, was collected from August to November 2009. Note that both data sets do not overlap, in the sense that, the code snippets in the two sets are distinct. During the sample collection phase, a custom Microsoft deobfuscator was used to identify obfuscation attempts. Each dataset was filtered based on a list of signatures of known cases. This gave us the positive (malicious) and negative (benign) labels needed for our evaluation. We extract the features for each snippet based on the alphabet of Table 1, and construct appropriate transactions of findings for each code snippet. The average size of transactions (i.e. the average number of findings per snippet) is also listed in Table 4.

The *Test Data* contains two snippets of Javascript that correspond to two new exploit attempts of a previously unknown Zero Day Vulnerability in Microsoft Internet Explorer – CVE-2009-3672. (See Section 2 for an example code snippet). These attempts first surfaced on 29, November 2009. Prior to this date, no signatures were available for these attacks (i.e., our signature scanners labelled them as benign). The CVE description for the Zero Day Vulnerability can be found at http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-3672 : *Microsoft Internet Explorer 6 and 7 does not properly handle objects in memory that (1) were not properly initialized or (2) are deleted, which allows remote attackers to execute arbitrary code via vectors involving a call to the getElementsByTagName method for the STYLE tag name, selection of the single element in the returned list, and a change to the outerHTML property of this element, related to Cascading Style Sheets (CSS) and mshtml.dll, aka "HTML Object Memory Corruption Vulnerability."* We show later in this section that our method was successful in detecting both the new exploits as attacks.

4.2 Training and validation

During the training phase, we used the *Train Data* a;gorithm to learn the model. The maximum size of patterns was fixed at 5 and the number of iterations for the EM algorithm was fixed at 100. The first step in our evaluation is to validate the stability of the classifier learnt. This is a key step in the model-building phase since, based on just the training data, we need to determine whether the training data was sufficient for the choice of parameters used. Validation involves randomly splitting the training set into 2 parts and using one part for learning the model, and the second part for evaluating performance under the learnt model. We expect that if the statistical estimation stabilizes, we should

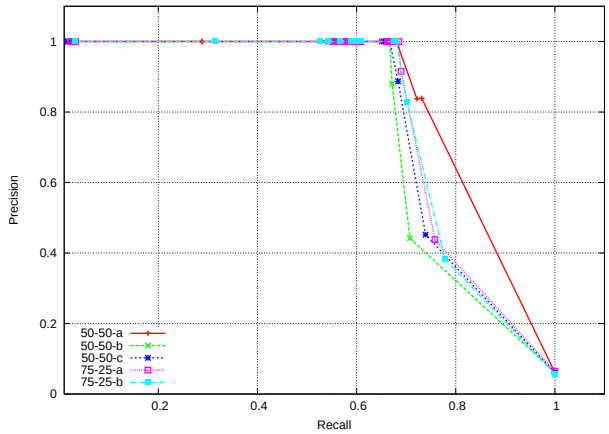

Figure 3: Cross validation results: Precision v/s Recall plots obtained by varying the classifier threshold for different splits of data into training and validation sets.

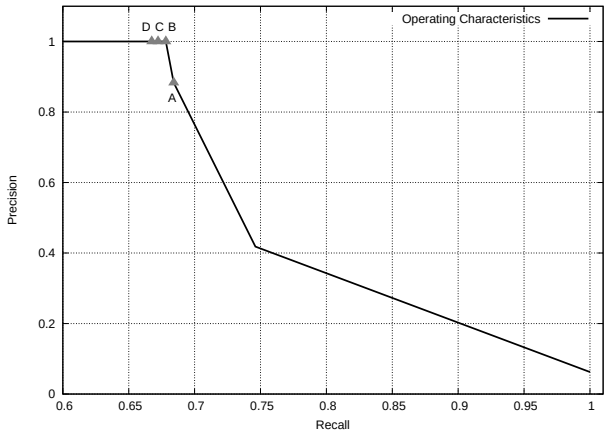

Figure 4: Finding an operating point for the classifier

Operating Point	# Malicious Cases Reported	# Known Issues based on Sigs	ZDV-I	ZDV-II
A	5991 (22%)	286 (1.0%)	√	√
B	624 (2.3%)	240 (0.8%)	√	√
C	614 (2.2%)	237 (0.8%)	√	×
D	560 (2.0%)	232 (0.8%)	√	×

Table 5: Results for detection of Zero Day Vulnerabilities in Test Data

get comparable results on different instances of random splitting of training data.

In our validation experiments, we generate 3 instances of random 2-way 50-50 splits and 2 instances of random 2-way 75-25 splits of the *Train Data* (These are referred to as 50-50-a, 50-50-b, 50-50-c, 75-25-a, 75-25-b and 75-25-c in the figures). In all the instances, the first parts (of size 50% or 75%) were used to train the corresponding models and the second parts (of size 50% or 25%) were used for validation. The cross-validation results are plotted in Fig. 3 in the form of precision v/s recall graphs. The different points in each curve correspond to the use of different thresholds on the likelihood ratios computed in line 4, *Algorithm 2*. The main observation is that the precision v/s recall behavior for all the instances are very similar – 100% precision can be achieved with a recall in the 65%-70% range. Then, as the threshold is relaxed (reduced) the algorithm will report more and more code snippets as malicious, thereby descreasing the precision of the classifier decision. The interesting region in the operating characteristics of Fig. 3 is at a precision of 90% and a recall of 70%, at the knee of operating curve (inflexion point). Moving substantially away from this knee would impact both precision and recall. Observe that operating points which achieve high precision (even if at the cost of some recall) are more important than those that achieve high recall (at the cost of precision). A high precision ensures that security experts do not have too many suspicious reports to analyze by-hand. Moreover, since benign cases far exceed malicious cases, a high recall result (at the cost of lower precision) is trivial and useless to the security expert looking for new ZDV exploits. In view of this, our result of 90% precision at 70% recall is very effective from the point-of-view of detecting new ZDV exploits. The important empirical result here is that such operating points were available (and identifiable) in all the instances we tested.

Finally, we plot the operating characteristic of the classifier using all of the *Train Data* in Fig. 4 – 100% of the data was used to learn the model and the same data was used for plotting the operating curve as well. The general behavior is similar to the plots in Fig. 3 (with marginally better precision since we are using all of the training data). We now select some suitable operating points for our classifier – in Fig. 4, these points are marked A, B, C and D. We basically pick 4 points in and around the 'knee' of the operating curve. Each operating point corresponds to a threshold to be used for the likelihood ratios during testing, In particular, the thresholds for operating point A was 1.59, for B was 2.31, for C was 3.02 and for D was 3.74. In the next subsection, we demonstrate the performance of our classifier at these four points.

4.3 Detecting Zero Day Vulnerabilities

Table 5 shows the results obtained on the test data. The first column mentions the operating point. The second column quotes the number of malicious cases reported for the corresponding operating point. The third column quotes the number of issues (amongst those reported) that were regarded as known issues based on a list signatures. The fourth and fifth columns indicate whether the algorithm detected the two instances of the ZDV exploit in the *Test Data*. As can be seen from the table, the first attack scenario (ZDV-I) was detected at all the operating points, while the second scenario (ZDV-II) was only detected at A and B (but not at C or D). In all cases except operating point A, the number of malicious cases reported is small (just around 2% of 26932 cases). This demonstrates that while the method does not report too many points as malicious, the ZDV cases almost always remain at the top of the output list (within 1 and 2% at a reasonable operating point).

4.4 Discussion

In the absence of ground truth, i.e., since we do not know if our test data has more ZDVs in it until it is disclosed, we cannot claim to be complete. Nevertheless, every transaction that has a high rank according to our algorithm, and is not filtered out by our signature scanners has the potential to be a ZDV and is worth further investigation. The robustness results from our cross-validation experiments, as well as the discovery of a real ZDV in our data underline the utility of our methods, and offer hope.

As we point out that our methods are sensitive to the choice of the features in our alphabet, and in the mix of features used for classification. The features we selected in our training dataset, all occur frequently in the ZDVs disclosed on the ZDI page. In general, our alphabet will also need to be updated when new methods of delivering attack payload are discovered, and we will need to retrain our models periodically.

Finally we observe that our methods have robustness built-in, evidenced by the discovery of two slightly different attack payloads that contained our ZDV. An adversary who uses a mix of obsfuscation techniques will have to work very hard to evade complete detection.

5. RELATED WORK

In this section we present related research in the context of data mining for security, and Javascript attack detection and prevention. Machine learning and data mining techniques have been applied to various security problems [11], including anomaly detection, intrusion detection, attack graph analysis, and analyzing audit trails for root kits, etc. A comprehensive discussion of these techniques and their applications is beyond the scope of this paper. The techniques we use in this paper, including frequent itemset mining [1] and our generative models [7] were presented earlier, while the learning component, with the expectation maximization, was developed for this tool.

In terms of other related work, in Argos [14] the authors present an x86 emulator for capturing and fingerprinting disclosed ZDVs, to provide accurate input to signature scanners. The Zero Day Initiative [13] maintains a current list of known ZDVs, which provides economic incentives to anyone reporting a new ZDV.

Researchers have studied a variety of static and semi-static techniques to address the misuse of Javascript language features that are used to exploit client-side application vulnerabilities, as well as mount attacks against web applications. We highlight some recent research in this context. In Blueprint [9], the authors propose an enhanced parser that needs to be installed on both clients and servers, which uses annotations to communicate an application developers intention to a client browser, with regard to where dynamic code can be downloaded. This effectively prevents unauthorized download, and an integrity checking function validates input to prevent code downloads. However, this solution requires changes to both servers and clients.

There are also techniques that only require changes on the client browsers. In their work on staged information flows [3], the authors present a combined static-dynamic technique that checks downloaded code against specified integrity and confidentiality requirements, and generates residual checks that can be validated at runtime. Techniques such as Gatekeeper [6] and [10] concentrate on smaller safer subsets of Javascript, rewriting code or adding wrappers, by analyzing malicious code on the victim machine. In Nozzle [15], heapsprays in particular are identified using a lightweight interpreter, and malicious and benign code are classified by characterising valid and invalid opcodes. In [4], the authors attach labels on sensitive information and identify malicious code that attempt to write, over-write or send these objects over a network, in browser extension code.

While many of these techniques are very effective in practice, a barrier to their adoption is the need to change browser or server code, or install new software on the clients. Our work is complementary to these efforts, as we are looking for the vulnerabilities that are being tested by the Javascript payload, and as long as unpatched machines exist, the attacks will continue. Further, Javascript is not the only vehicle for delivering ZDVs, e.g., corrupt video or image files, and Microsoft word macros can all carry attack payload, and our technique can be adopted to look for findings in these files.

6. CONCLUSIONS

Detecting Zero Day Vulnerabilities is a critical problem that confronts security response teams on an everyday basis. Malware writers are on the constant look-out for exploits of new (previously unknown) vulnerabilities, exploits which are designed to trick standard signature-based malware detection techniques. In this paper, we develop methods, which have the potential for detecting the new vulnerabilities even before they are successfully exploited in the field. Our methods are based on a rigorous statistical characterization of previously known exploits. We use a combination of frequent pattern mining techniques from data mining and generative model estimation techniques from machine learning to develop a statistical filter for detecting malicious payload intended to exploit a new vulnerability. Our results demonstrate that our techniques are robust and are able to detect high-value ZDVs in real-world data sets. Although this paper primarily focusses on detecting ZDVs in Javascript payload, we expect that our methods are equally applicable in several other contexts including malicious audio and video files, Microsoft Word and Excel macro viruses, etc.

7. REFERENCES

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 207–216, May 1993.

[2] J. Bilmes. A gentle tutorial on the EM algorithm and its application to parameter estimation for gaussian mixture and hidden markov models. Technical Report TR-97-021, International Computer Science Institute, Berkeley, California, Apr. 1997.

[3] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for javascript. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, New York, NY, USA, 2009. ACM.

[4] M. Dhawan and V. Ganapathy. Analyzing information flow in javascript-based browser extensions. In *ACSAC'09: Proceedings of the 25th Annual Computer*

Security Applications Conference, pages 382–391, Honolulu, Hawaii, USA, December 2009. IEEE Computer Society Press, Los Alamitos, California, USA. http://dx.doi.org/10.1109/ACSAC.2009.43.

[5] J. Franklin, V. Paxson, S. Savage, and A. Perrig. An inquiry into the nature and causes of the wealth of internet miscreants. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 375–388, New York, NY, USA, 2007. ACM.

[6] S. Guarnieri and B. Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code. In *Proceedings of the Usenix Security Symposium*, Aug. 2009.

[7] S. Laxman, P. Naldurg, R. Sripada, and R. Venkatesan. Connections between mining frequent itemsets and learning generative models. In *Proceedings of the Seventh IEEE International Conference on Data Mining ICDM 2007*, pages 571–576, Omaha, Oct. 2007.

[8] S. Laxman, V. Tankasali, and R. White. Stream prediction using a generative model based on frequent episodes in event sequences. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'08)*, pages 453–461, Las Vegas, Aug. 2008.

[9] M. T. Louw and V. N. Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *IEEE Symposium on Security and Privacy*, pages 331–346. IEEE Computer Society, 2009.

[10] S. Maffeis and A. Taly. Language-based isolation of untrusted javascript. In *CSF '09: Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium*, pages 77–91, Washington, DC, USA, 2009. IEEE Computer Society.

[11] M. A. Maloof. *Machine Learning and Data Mining for Computer Security: Methods and Applications (Advanced Information and Knowledge Processing)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[12] C. Miller. The legitimate vulnerability market: Inside the secretive world of 0-day exploit sales. In *In Sixth Workshop on the Economics of Information Security*, 2007.

[13] T. Point. The zero day initiative.

[14] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. *SIGOPS Oper. Syst. Rev.*, 40(4):15–27, 2006.

[15] P. Ratanaworabhan, B. Livshits, and B. Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *Proceedings of the Usenix Security Symposium*, Aug. 2009.

[16] SANS. The top cyber security risks 2009, Sept. 2009.